A METHOD OF DETERMINING THE SYNTACTIC CORRECTNESS OF EXPRESSIONS

Technical Field of the Invention

The present invention relates to the determination of the syntactic correctness of expressions, such as those used in computer programs.

Background Art

Algebraic expressions are typically used in computer programs to assign values to variables. These expressions normally occur on the right hand side of an assignment statement, and in particular after an assignment operator. The assignment operator most often used is an "=" sign. They are also used in parameter lists of functions, in conditional statements, etc., in computer programs. An example of an assignment statement is:

```
S = a + b*c + fn(a, b+c, d) + func(a, fn(c,d,e))        (1)
```

wherein variable `S` is assigned a value, the value being dependent on `a`, `b`, `c`, `d` and `e`, each of which is either a number or a variable, and fn() and func(), which are functions.

Prior methods of determining the syntactic correctness of expressions routinely break up the expression on the right hand side of the assignment operator into tokens and create a tree, where the tokens sit at the nodes of the tree. A number of rules of how to deal with those tokens, that have been created, are then applied.

Disclosure of the Invention

It is an object of the present invention to provide an alternate and/or improved method of determination of the syntactic correctness of expressions.

According to a first aspect of the invention, there is provided a method of determining the syntactic correctness of an expression for use in a computing environment, said method comprising the steps of:

(a) creating a string of characters from said expression;

(b) iteratively substituting occurrences of said characters also occurring in a first predetermined list with characters from a second predetermined list; and

(c) determining said expression to be syntactically correct only if said string reduces to a single predetermined character.

Preferably, characters not permitted in said string are substituted with a special character from said second predetermined list that causes said iteration to be ceased. Alternatively, these characters are substituted with a special character from said second predetermined list that causes said expression to be determined to be syntactically incorrect on completion of said iterative substitution step. Preferably, said expression is algebraic, and the creation step includes substituting each variable and variable operators with a single variable character, and retaining algebraic characters. In an alternative embodiment, said expression is a filename, and the creation step includes substituting subdirectory names, filenames and filename extensions with a single character, and retaining delimiter characters. In yet another embodiment, said expression is a variable or function name, and the creation step includes substituting a number with one character, and substituting an alphabet or an underscore with a second character.

According to another aspect of the invention there is provided a computer readable memory medium having recorded thereon a computer program for implementing the method described above.

The present invention deals with the entire expression in its operations. It essentially mimics the way humans visually interpret and check an expression's syntax. Thus the invention looks for substrings (character sequences), which are legitimate and replaces them with a shorter but semantically equivalent substring. It also looks for character sequences which are illegitimate, and replaces them with a substring (i.e. "?") to indicate that it has met with an illegal sequence. It does these operations repeatedly, and in predetermined sequences, till no further changes in the string can be made; that is, the string achieves a constant length.

Since the invention does not rely upon operator and operand tokens, but on character combinations appearing in a character string, it is not limited to determining the

syntactic correctness of algebraic expressions alone but can also be used to determine the syntactic correctness of other types of character strings. These may include filenames and variable names, provided the syntax rules of their formation are known. Such rules must also be consistent and unambiguous.

Brief Description of the Drawings

A number of preferred embodiments of the present invention will now be described with reference to Fig. 1, in which is a flow diagram of a method of determining whether an algebraic expression is syntactically correct.

Detailed Description including Best Mode

Referring to Fig. 1, a flow diagram is shown of a method 100 of determining whether an algebraic expression is syntactically correct. Starting at step 10, a set of characters is defined called a delimiter character set. The delimiter character set is a designated set of special characters, which act as markers within a character string `strg`, created from the algebraic expression. Character(s) of the character string `strg` appearing between such markers, or between the beginning of the character string `strg` and a marker, or between a marker and the end of the character string `strg` is/are treated as a single entity. If the delimiter set is empty, then each of the characters in the algebraic expression is treated as an entity.

Entities belong to a certain predefined set of types. In the algebraic expression they are generically called a variable `"v"`. In a filename they are called a filename component `"f"`, whereas in a variable name they are called an alphabet `"a"` component if the entity is an alphabet or an underscore, and a number `"n"` if it is a number. If the entity cannot be matched to a predefined type, it is typed `"?"`.

The delimiter character set is compactly expressed as a string, where each character within the quotation marks is a delimiter character. An example of a delimiter character set for an algebraic expression is "+-*/,()".

Step 20 defines a `FromStrg[]` and a `ToStrg[]` string array. There is a one-to-one correspondence between the elements of the two arrays, `FromStrg[]` and

ToStrg[], such that the i-th element in the FromStrg[] array, corresponds to the i-th element in the ToStrg[] array. The number of elements in the FromStrg[] array are therefore equal to the number of elements in the ToStrg[] array. The NULL string at the end of each array FromStrg[] and ToStrg[], is used to indicate the end of the array.

The FromStrg[] string array contains a list of character combinations which comprehensively describe the combinations which can be detected to be either correct or incorrect. The ToStrg[] string array contains a corresponding list of character combinations, such that if character string strg contains the i-th element from the FromStrg[] string array, it can be replaced by the corresponding i-th element from the ToStrg[] string array. The task of such replacements is to progressively simplify the string strg and bring it to the type it is supposed to represent. An element in the ToStrg[] string array will always have a lesser number of characters than its corresponding element in the FromStrg[] string array, since the element in the ToStrg[] string array is expected to represent a simplified version of the corresponding element in the FromStrg[]. An illegal element in the FromStrg[] string array will have a "?" element as its corresponding element in the ToStrg[] string array. A legal element in the FromStrg[] string array will have a more cryptic but semantically correct combination of delimiter and/or type characters.

Often a one-step replacement between an element in the FromStrg[] string array, consisting of delimiter and type characters, to a legitimate element in the ToStrg[] string array, also consisting of delimiter and type characters, may not be possible (or obvious). In such cases multiple step replacements are used. One or more combinations participating in such multistep replacements will carry additional characters, such as an "#" character. Such characters play the role of temporary intermediaries and may be called intermediate characters.

With respect to the delimiter character set defined for an algebraic expression, the two string arrays are defined as (in the notations of C programming language):

```
FromStrg[] = {")(",    "(v)()",   "(v)(v)",   "v(v)",   "v(#)",   "v()",
              "(v)",   "vv",      "v[v]",     "(v,v)",  "(+v)",   "(-v)",
              "v*v",   "v/v",     "v+v",      "v-v",    ",v,",    ""};
```

```
ToStrg[] =   {"?",   "?",   "?",   "v",   "v",   "v",
              "v",   "?",   "v",   "(#)", "(v)", "(v)",
              "v",   "v",   "v",   "v",   ",",   ""};
```

In step 30, the character string `strg` is created from the algebraic expression being analysed. Thus for a given character string, the derived character string `strg` obtained by step 30 will be a sequence of delimiter characters and type characters. Step 30 is performed by the following substeps:

In substep 31 (not shown), it is determined whether the given algebraic expression begins with a unitary + or − operator. If this is so, then the expression is prefixed with the numeral `0`. Alternatively, the algebraic expression may be enclosed in brackets. Furthermore, all blank (or space) characters are removed from the algebraic expression. A counter variable `i` is initiated to `0`.

In substep 32 (not shown), the expression is scanned from left to right, character by character, until a delimiter character is found or the end of the expression is reached. A variable delimiter character α is set equal to the delimiter character found.

(a) If no characters are found before the delimiter character α, then the i-th character in the character string `strg` is set equal to the delimiter character as follows: `strg[i] =` α. The counter variable `i` is also incremented by 1 and the procedure continues to substep 33 (not shown).

(b) If one or more characters are found before the delimiter then it is determined whether it/they form(s) a valid variable name, function name or number. If the character(s) is/are valid, then a character "v" is put into string position `strg[i]`. Alternatively a character "?' is put into string position `strg[i]`. The following string position in the string `strg[]` is filled with the delimiter character α as follows: `strg[i+1] =` α. The counter variable `i` is increased by 2 and the procedure continues to substep 33.

Substep 33 determines whether the end of the expression has been reached. If this is not so, then the procedure continues to substep 32, assuming that the expression now begins at the character immediately following the delimiter character α.

As an example, the algabraic expression obtained from the assignment statement (1) is as follows:

`a + b*c + fn(a, b+c, d) + func(a, fn(c,d,e))` (2)

and the following string `strg[]` will be produced by following substeps 31 to 33:

`v+v*v+v(v,v+v,v)+v(v,v(v,v,v))` (3)

Step 40 initialises a variable `size` as the length of the string `strg`, including an end of string character.

The method 100 continues to step 50 where it is determined whether the constructed string `strg[]` contains one or more "?" characters. If the string `strg[]` contains at least one "?" character, then the algebraic expression is syntactically incorrect, and the method 100 ends in step 90 by returning that the expression is not correct. Alternatively, the method 100 continues to step 51. Step 51 sets the counter variable `i` equal to `-1`, while step 52 increments the counter variable `i` by 1.

Step 53 determines whether the counter variable `i` is still smaller than `n`, where `n` is the dimension of the `FromStrg[]` and `ToStrg[]` arrays. If this is affirmative, then step 54 replaces each occurrence of the string `FromStrg[i]` in string `strg` with `ToStrg[i]`. Steps 52 to 54 are repeated until step 53 determines that the end of the `FromStrg[i]` array has been reached.

The method 100 then continues to steps 55 and 56 where it is determined whether the resulting string `strg` is the same size as before steps 51 to 54, by comparing it with the variable `size`. If the length of the string `strg` has changed, then the method 100 returns to step 50 after the variable size has been set to the new length of the string `strg`.

If the length of the string `strg` has not changed, it means that the string `strg` has been reduced as far as possible and the method 100 continues to step 58 where it is determined whether the string `strg` equals the character "`v`". Only if `strg="v"` is the algebraic expression syntactically correct and the method 100 ends in step 80. (The significance of this is that any expression must eventually be reducible to a number or a variable.) Alternatively the algebraic expression is syntactically incorrect, and the method 100 ends in step 90 by returning that the expression is not correct.

As an illustration of steps 40 onwards described above, consider the further processing of the example string `strg` of equation 3, the method 100 commencing from step 40 where it is determined that the variable `size` equals 31, and after a first iteration repeating steps 52 to 54, the string `strg` is reduced to:

`v(v,v)+v(v,v(v,v))` (4)

Because the variable `j`, determined in step 55 is equal to 19, which step 56 determines not to be equal to the variable `size`, the variable `size` is therefore set to 19 in step 57 and the method 100 proceeds to step 50. The iteration in steps 52 to 54 is repeated and the string `strg` is reduced to:

`v(#)+v(v,v(#))` (5)

Again steps 55 and 56 determine that the string `strg` has reduced and another iteration of steps 52 to 54 further reduces the string `strg` to:

`v(#)` (6)

Yet a further iteration of steps 52 to 54 reduces the string `strg` to:

`v` (7)

Steps 55 and 56 again determine that the string `strg` has reduced and the method 100 proceeds to step 50 after the variable `size` has been set to 2 in step 57. However, the iteration of steps 52 to 54 does not further reduce the string `strg` and step 56 therefore directs the method 100 to step 58. Step 58 determines that the string `strg` is

equal to the string "v", and the expression of equation 2 is therefore determined to be syntactically correct.

Examples of algebraic expressions that are not syntactically correct are:

a*+b	: there are two operators between variables a and b;
*a*b	: an expression can not start with a multiplication operator;
a(b+c)	: there is no operator between the variable a and the operator (;
a*(b+c))	: the last) operator does not have a matching (operator.

Source or psuedo-code for performing steps 50 to 90 of the method 100 is as follows:

```
// strg[] is the character array derived from the given expression
// using substeps 31-33 described above.
// size is the size of the string strg.
// n is the dimension of FromStrg[] and ToStrg[] arrays.
// ChangeSubstrg() replaces FromStrg[i], if found in strg with ToStrg[i].
// cond is a boolean flag which saves the result of the iteration.
// It is TRUE if the strg is syntactically correct, else FALSE.

size = strlen(strg) + 1;
cond = TRUE;
while (cond) {
  if(strchr(strg,(int)'?')break;
  i = -1
  while (++i < n)  {
     while (strstr(strg, FromStrg[i]))
        ChangeSubstrg(strg, FromStrg[i], ToStrg[i];
  }
  j = strlen(strg) + 1;
  if (size == j) break;
  size = j;
 }
if (strcmp(strg, "v") != 0) cond = FALSE;
```

```
if (cond) Msg("Expression is correct");
else Msg("Expression is incorrect");
```

The method 100 may be viewed as adding to the list of isxxx() functions typically found in modern compiler libraries, such as (in C), isalpha(), isdigit(), etc. The present method 100 may be encoded into an isexpression() function.

In an alternative embodiment, the method 100 is used to determine the syntactic correctness of a filename. For example, suppose a filename can have one of the following three syntaxes:

(i) `<drive name>:/<dir name>/<subdir name>/…/<subdir name>/<filename>/.<ext>`

(ii) `<filename>.<ext>`

(iii) `<filename>`

Following the method 100, and in particular step 10, a delimiter character set is defined as "./". Step 20 defines the following string arrays:

```
FromStrg[] = {"f.f.f", "f.f/", "f.f", "/f/", "f:", "/#", "#/f", ""}
```

and

```
ToStrg[]   = {"?",     "?",    "f",   "/",   "#",  "?",  "f",   ""}
```

Here `f` represents a (sub)directory name, filename or a filename extension. Hence if filename is syntactically correct, it will initially produce the string "`f:/f/f/…/f/f.f`", "`f.f`" or "`f`" respectively, for the syntaxes (i), (ii), (iii) noted above.

Only if the filename is syntactically correct, then a string "`f`" will result from the method 100. Step 58 determines whether the string `strg` has been reduced to the character "`f`".

This embodiment may be coded as an IsFilename() function, which takes a character string, presumed to be a filename, as its input.

In yet another embodiment, the method 100 is used to determine whether an expression forms a valid variable or function name. It is assumed that this expression must start with an alphabetical character or an underscore, followed by a sequence of characters, each of which can be an alpha-numeric character or an underscore.

In step 30, scanning the expression string from left to right character by character, if the character is a number, it is replaced with the character "`n`". Similarly, if the character is an alphabetical character or an underscore, it is replaced with the character "`a`". If the character is anything else, then it is replaced with the character "`?`".

In this embodiment, the delimiter character set defined is empty, while the following two string arrays are defined in step 20:

```
FromStrg[] = {"aa", "an", ""}
```
and
```
ToStrg[] = {"a", "a", ""}
```

Only if the string expression is a variable name will step 58 determine that the string `strg` has reduced to the character "`a`" and proceed to step 80 for correct expressions.

This embodiment can be coded as an IsVar() function, which takes the character string, presumed to be a variable, as its input.

Embodiments of the invention can be implemented within compilers, for example. As is well known, a compiler generates machine executable object code from high-level source code, written in languages such as C++ and Java™.

The foregoing describes only some embodiments of the present invention, and modifications and/or changes can be made thereto without departing from the scope and spirit of the invention, the embodiments being illustrative and not restrictive.